A Component Model for Distributed Embedded Real-Time Systems

Uwe Rastofer
Frank Bellosa

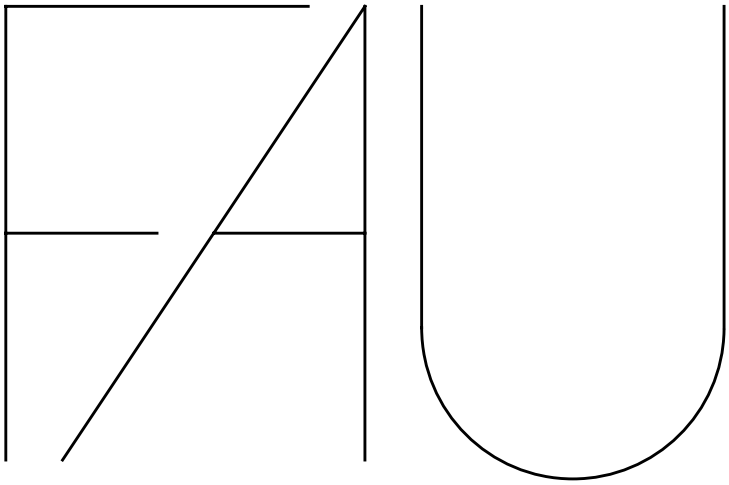
September 1999 TR-I4-99-01

Technical Report

Department of
Computer Science

Operating Systems — IMMD IV

Friedrich-Alexander-University
Erlangen-Nürnberg, Germany



This work was presented at the GCSE'99 Young Researchers Workshop which was part of the First International Symposium on Generative and Component-Based Software Engineering (GCSE'99).

A Component Model for Distributed Embedded Real-Time Systems

- Work in Progress -

Uwe Rastofer, Frank Bellosa

Department of Computer Science IV, University of Erlangen-Nürnberg, Martensstraße 1, 91058 Erlangen, Germany

Uwe.Rastofer@informatik.uni-erlangen.de
Frank.Bellosa@informatik.uni-erlangen.de

1. Introduction

Currently popular component models like Microsoft's COM [EdEd98] and Sun's Java Beans [Eng97] rely on a more or less uniform execution platform. It is assumed that on this platform a component can use nearly unlimited virtual memory, that there are no other time constraints apart from the user's patience, and that one component's resource usage will not have any influence on other components. So these component models are only concerned with functionality and do not explicitly address non-functional issues like the usage of resources.

These assumptions do no longer hold in the field of distributed embedded real-time systems. There is a variety of different hardware architectures and operating systems on the embedded market which creates a large number of diverse execution platforms. Resources like processor time and memory are scarce and one component can easily influence others by consuming too many resources. Distribution issues play an important role as well. Execution times are much larger when a component on different node has to be contacted.

So on the one hand, to produce a correctly working embedded real-time system all properties of the execution environment have to be taken into account and the components have to be adapted to this environment. On the other hand, the (base) functionality of these components is still independent of a special execution environment. To overcome platform-specific components we propose a component model that separates the component's functionality from platform-specific issues like concurrency, synchronization, and distribution. Components are developed in a platform-independent model and are later mapped to execution environments that introduce platform-specific features.

2. The Platform-Independent Component Model

In the platform-independent model components communicate by sending uni-directional events. Events are typed and they may contain data. An event can be received by a component only through an input port that has a compatible type. The reception of an event induces an activity in the receiving component. Otherwise components are passive and have no activity of their own. The induced activity executes an event handler that is associated with this input port. During the execution of the event handler new events can be sent to the component's output ports. It is the responsibility of the output port to distribute an event to all connected input ports of other components. When the event handler returns the activity that was injected into the component also terminates, i.e. event handlers have run-to-completion semantics.

In principle, a component can receive any number of events at the same time and therefore many event handlers may be executed concurrently. When event handler access shared data these accesses have to be synchronized. As shown in [Rei98] synchronization can be separated from the algorithmic code by attaching special objects that encapsulate the synchronization strategy. Each synchronization object can either implement one basic synchronization strategy or can be built from a set of basic synchronization objects. In our model synchronization takes place when a component receives an event on an input port. So each synchronized component has to specify a synchronization strategy and a mapping of its ports to that strategy.

An embedded application that was developed using our component model consists of a directed graph of connected components. Since all components are passive there is no internal activity inside the application. Instead the application is activated by external events that are created by the environment, e.g. in response to a hardware interrupt or an expiring timer. Each external input event generates a chain of internal events when it is propagated through the graph of components.

Ultimately these events will generate external output events that are consumed by the environment. The application developer can annotate such event chains with timing information that is essential for the correct behavior of the embedded system. Like in [Cor98] the timing information shows end-to-end constraints for related input and output events.

3. Mapping to an execution environment

When an embedded application is mapped to its execution environment the graph of connected components is first of all partitioned into set of related components. Each set is then mapped to a single node of the distributed execution environment. All external input events of components in a set must be generated in the form of tasks on the node that the set is mapped to. In addition all events that are received from other nodes must also create local tasks. Some of the external events can only be generated or consumed on a special node which creates location constraints for some components.

After the location of a component is known the execution times of its event handlers can be estimated or measured. From the event handler's execution times the execution time for a chain of events can be computed.

The next step is to assign tasks to chains of events. Although it would be possible to create a new task for each event handler activation this is highly inefficient and can be avoided, e.g. when only one output event is generated by this event handler. For these tasks a schedule has to be created. This schedule must also take into account the synchronization requirements of the components. Finally when a schedule has been found, localized versions of the components are generated. These localized versions contain the event handlers, synchronization code, and code to implement the task schedule.

4. Summary

In distributed embedded real-time systems the execution platform has a huge influence on non-functional properties of components. Therefore we propose a component model that separates the component's functionality from the platform-specific issues concurrency, synchronization, and distribution. The resulting components can more easily be adapted to the constraints of the execution environment.

References

Cor98 P. Cornwell: Reusable Component Engineering For Hard Real-Time Systems. Ph.D. dissertation, University of York, UK, 1998

EdEd98 G. Eddon, H. Eddon: Inside Distributed COM. Microsoft Press, 1998.

Eng97 R. Englander: Developing Java Beans. O'Reilly, Sebastopol, 1997.

Rei98 S. Reitzner: "Virtual Synchronization: Uncoupling Synchronization Annotations from Synchronization Code". In: Proceedings of the ACM SAC'98, Symposium on Applied Computing, New York, 1998